Science of Computer Programming 6 (1986) 135-157
North-Holland

A LARCH SHARED LANGUAGE HANDBOOK*

J.V. GUTTAG
MIT Laboratory for Computer Science, Cambridge, MA 02139, U.S.A.

J.J. HORNING
Digital Equipment Corporation, Systems Research Center, Palo Alto, CA 94301, U.S.A.

Communicated by M. Sintzoff
Received July 1984
Revised March 1985

Abstract. This handbook consists of a collection of traits written in the Larch Shared Language, and is intended as a companion to the "Report on the Larch Shared Language". It should serve three distinct purposes:
- Provide a set of components that can be directly incorporated into other specifications;
- Provide a set of models upon which other specifications can be based; and
- Help people to better understand the Larch Shared Language by providing a set of illustrative examples.

Preface

This Handbook consists of a collection of traits written in the Larch Shared Language, and is intended as a companion to the "Report on the Larch Shared Language" [3]. It should serve three distinct purposes:
- Provide a set of components that can be directly incorporated into other specifications;
- Provide a set of models upon which other specifications can be based; and
- Help people to better understand the Larch shared language by providing a set of illustrative examples.

We have tried to isolate the 'smallest useful increments' of specification that it might be reasonable to use in other specifications. In particular, we have tried to provide traits that will make it convenient to specify the weak assumptions that characterize many of the more widely applicable specifications. This is particularly evident in the "Container properties" and "Container classes" sections. The traits in these sections are smaller and more numerous than is typical in 'from scratch' specifications. This sometimes leads to a somewhat overstructured appearance.

* This work was supported at MIT's Laboratory for Computer Science by DARPA under contract N00014-75-C-0661, and by the National Science Foundation under Grant MCS-811984 6, by the Digital Equipment Corporation Systems Research Center, and at the Xerox Palo Alto Research Center by the Computer Science Laboratory.

In addition to traits that we expect to be directly incorporated in specifications, we have included a number of traits intended primarily as patterns. The "Generic operators on containers" section contains several such traits. Specifiers are more likely to edit these traits than to include them, because they will need similar operators with different arities.

We have mostly stuck to familiar examples. Since they describe well-understood mathematical entities, many of the traits, e.g., Integer, are atypically complete. In general, we expect most specifications to supply constraints, rather than complete definitions. The "Display traits" section is more typical in this respect.

The support tools envisioned for Larch are not yet available. Transcriptions of traits in this paper have been mechanically checked for some properties. Several errors were found and corrected as a result of this checking, but others may not have been detected and some additional transcription errors may have crept in. Thus these traits should be given the same sort of credence as carefully written programs that have not been checked by a compiler.

We would like to be able to present specifications with the clarity and rigor of a mathematics text, as advocated in [1]. In particular, the formal text should be accompanied by a substantial amount of informal commentary. However, the present Handbook contains only the formal material, and corresponds more nearly to an appendix of 'collected formulas' than to a text.

Conventions

The identifier T is used to identify the only interesting sort in generic traits.
The identifiers C and E are used for 'containing' and 'element' sorts.
The infix symbol $\# \bigcirc \#$ is used to denote a generic binary operator.
The infix symbol $\# \circledR \#$ is used to denote a generic relational operator.
Asserts is used rather than **constrains** when constrains would supply no information (e.g., because there is only one operator).

Basic properties of single operators

Associative: **trait**
introduces $\# \bigcirc \#$: T, T→T
asserts for all [x, y, z: T]
$(x \bigcirc y) \bigcirc z = x \bigcirc (y \bigcirc z)$

Commutative: **trait**
introduces $\# \bigcirc \#$: T, T→Range
asserts for all [x, y: T]
$x \bigcirc y = y \bigcirc x$

Idempotent: **trait**
introduces op: T→T
asserts for all [*x*: T]
op(op(*x*)) = op(*x*)

Involutive: **trait**
introduces op: T→T
op(op(*x*)) = *x*

Basic properties of binary relations

Relation: **trait**
introduces # ® #: T, T→Bool

TotalRelation: **trait**
includes Relation
asserts for all [*x*, *y*: T]
(*x* ® *y*)|(*y* ® *x*)

Reflexive: **trait**
includes Relation
asserts for all [*x*: T]
x ® *x*

Irreflexive: **trait**
includes Relation
asserts for all [*x*: T]
¬(*x* ® *x*)*

Transitive: **trait**
includes Relation
asserts for all [*x*, *y*, *z*: T]
((*x* ® *y*) & (*y* ® *z*))⇒(*x* ® *z*)

ReflexiveTransitive: **Trait**
includes Reflexive, Transitive

Symmetric: **trait**
includes Relation
asserts for all [*x*, *y*: T]
(*x* ® *y*) = (*y* ® *x*)
implies Commutative **with** [® **for** ◯, Bool **for** Range]

Equivalence: **trait**
includes ReflexiveTransitive **with** [.eq **for** ®],
Symmetric **with** [.eq **for** ®]

Ordering relations

PartialOrder: **trait**
imports ReflexiveTransitive **with** [$\leq$ **for** ®]

TotalOrder: **trait**
includes PartialOrder, TotalRelation **with** [$\leq$ **for** ®]

OrderEquivalence: **trait**
assumes PartialOrder
introduces #.eq#: T, T → Bool
constrains .eq **so that for all** [x, y: T]
$(x, .\text{eq}\ y) = (x \leq y)\ \&\ (y \leq x)$
implies Equivalence
converts [.eq]

OrderEquality: **trait**
assumes PartialOrder
includes Equality, OrderEquivalence **with** [= **for** .eq]

PartialOrderWithEquality: **trait**
includes PartialOrder, OrderEquality

TotalOrderWithEquality: **trait**
includes TotalOrder, OrderEquality

DerivedOrder: **trait**
assumes PartialOrder
introduces
$\# < \#$: T, T → Bool
$\# \geq \#$: T, T → Bool
$\# > \#$: T, T → Bool
constrains $<$ **so that for all** [x, y: T]
$(x < y) = ((x \leq y)\ \&\ (\neg(y \leq x)))$
constrains $\geq$ **so that for all** [x, y: T]
$(x \geq y) = (y \leq x)$
constrains $>$ **so that for all** [x, y: T]
$(x > y) = (y < x)$
implies Transitive **with** [$<$ **for** ®],
Transitive **with** [$>$ **for** ®],
PartialOrder **with** [$\geq$ **for** $\leq$]
converts [$<$, $\geq$, $>$]

PartiallyOrdered: **trait**
  imports PartialOrderWithEquality
  includes DerivedOrders
  implies PartialOrderWithEquality **with** [$\geq$ **for** $\leq$]

Ordered: **trait**
  imports TotalOrderWithEquality
  includes DerivedOrders
  implies PartiallyOrdered, TotalOrderWithEquality **with** [$\geq$ **for** $\leq$]

Group theory

LeftIdentity: **trait**
  introduces
    # ◯ #: T, T→T
    unit: →T
  asserts for all [*x*: T]
    unit ◯ *x* = *x*

RightIdentity: **trait**
  introduces
    # ◯ #: T, T→T
    unit: →T
  asserts for all [*x*: T]
    x ◯ unit = *x*

Identity: **trait**
  includes LeftIdentity, RightIdentity

LeftInverse: **trait**
  assumes LeftIdentity
  introduces inv: T→T
  asserts for all [*x*: T]
    inv(*x*) ◯ *x* = unit

RightInverse: **trait**
  assumes RightIdentity
  introduces inv: T→T
  asserts for all [*x*: T]
    x ◯ inv(*x*) = unit

Inverse: **trait**
  assumes Identity
  includes LeftInverse, RightInverse

Abelian: **trait**
  imports Commutative **with** [T **for** Range]

Semigroup: **trait**
includes Associative, Equality

LeftMonoid: **trait**
includes Semigroup, LeftIdentity

RightMonoid: **trait**
includes Semigroup, RightIdentity

Monoid: **trait**
includes LeftMonoid, RightMonoid

Group: **trait**
includes LeftMonoid, LeftInverse
implies RightMonoid, RightInverse, Involutive **with** [inv **for** op]

AbelianSemigroup: **trait**
includes Abelian, Semigroup

AbelianMonoid: **trait**
includes Abelian, LeftMonoid
implies Monoid

AbelianGroup: **trait**
includes Abelian, Group

Distributive: **trait**
introduces
$\# + \#$: T, T → T
$\# * \#$: T, T → T
asserts for all [x, y, z: T]
$x * (y + z) = (x * y) + (x * z)$
$(y + z) * x = (y * x) + (z * x)$

Simple numeric types

Ordinal: **trait**
includes Ordered **with** [Ord **for** T]
introduces
first: → Ord
succ: Ord → Ord
asserts Ord **generated by** [first, succ]
Ord **partitioned by** [$\leq$]
for all [x, y: Ord]
first $\leq x$

```
        ¬(succ(x) ≤ first)
        succ(x) ≤ succ(y) = x ≤ y
    converts [=, ≤, <, ≥, >]

Cardinal: trait
    imports Ordinal with [0 for first, Card for Ord]
    introduces
        1: →Card
        # + #: Card, Card→ Card
        # * #: Card, Card→ Card
        # ⊖ #: Card, Card→ Card
    constrains 1 so that 1 = succ(0)
    constrains +, * so that for all [x, y: Card]
        x + 0 = x
        x + succ(y) = succ(x + y)
        x * 0 = 0
        x * succ(y) = x + (x * y)
    constrains ⊖ so that for all [x, y: Card]
        0 ⊖ x = 0
        x ⊖ 0 = x
        succ(x) ⊖ succ(y) = x ⊖ y
    implies
        Card generated by [1, +, ⊖]
        Card partitioned by [≥], by [=], by [<], by [>]
        for all [x, y: Card] x ≤ y = ((x ⊖ y) = 0)
        Cardinal2
    converts [1, ⊖, +, *, =, ≤, ≥, <, >]

Cardinal2: trait  % Alternate definition for comparison
    includes AbelianMonoid with [+ for ◯, 0 for unit, Card for T],
        AbelianMonoid with [* for ◯, 1 for unit, Card for T],
        Distributive with [Card for T],
        Ordered with [Card for T]
    introduces
        # ⊖ #: Card, Card→ Card
        succ: Card→ Card
asserts Card generated by [0, 1, +]
        for all [x, y: Card]
            x < (x + 1)
            (x + y) ⊖ y = x
            0 ⊖ x = 0
            succ(x) = x + 1
    implies Cardinal
```

Simple data structures

Pair: **trait**
introduces
⟨#, #⟩: T1, T2 → C
#.first: C → T1
#.second: C → T2
asserts C **generated by** [⟨#, #⟩]
C **partitioned by** [.first, .second]
for all [*f:* T1, *s:* T2]
⟨*f*, *s*⟩.first = *f*
⟨*f*, *s*⟩.second = *s*
implies converts [.first, .second]

Triple: **trait**
introduces
⟨#, #, #⟩: T1, T2, T3 → C
#.first: C → T1
#.second: C → T2
#.third: C → T3
asserts C **generated by** [⟨#, #, #⟩]
C **partitioned by** [.first, .second, .third]
for all [*f:* T1, *s:* T2, *t:* T3]
⟨*f*, *s*, *t*⟩.*first* = *f*
⟨*f*, *s*, *t*⟩.second = *s*
⟨*f*, *s*, *t*⟩.third = *t*
implies converts [.first, .second, .third]

FiniteMapping: **trait**
assumes Equality **with** [Index **for** T]
introduces
new: → C
bind: C, Index, E → C
#[#]: C, Index → E
defined: C, Index → Bool
asserts C **generated by** [new, bind]
C **partitioned by** [#[#], defined]
constrains C **so that**
for all [*c:* C, *i*, i_1*:* Index, *e:* E]
bind(*c*, i_1, *e*)[*i*] = **if** *i* = i_1 **then** *e* **else** *c*[i]
¬defined(new, *i*)
defined(bind(*c*, i_1, *e*), *i*) = (*i* = i_1) | defined(*c*, *i*)
implies converts [#[#], defined]
exempts for all [*i:* Index] new[*i*]

Container properties

Container: **trait**
introduces
new: →C
insert: C, E→C
asserts C **generated by** [new, insert]

Singleton: **trait**
assumes Container
introduces singleton: E→C
constrains singleton **so that for all** [*e:* E]
singleton(*e*) = insert(new, *e*)
implies converts [singleton]

IsEmpty: **trait**
assumes Container
introduces isEmpty: C→Bool
asserts for all [*c:* C, *e:* E]
isEmpty(new)
¬isEmpty(insert(*c*, *e*))
implies converts [isEmpty]

Size: **trait**
assumes Container
imports Cardinal
introduces size: C→Card
constrains size **so that**
size(new) = 0

AdditiveSize: **trait**
assumes Container
includes Size
constrains size, insert **so that for all** [*c:* C, *e:* E]
size(insert(*c*, *e*)) = size(*c*) + 1
implies converts [size]

Join: **trait**
assumes Container
introduces #.join #: C, C→C
constrains .join **so that for all** [*c*, c_1: C, *e:* E]
c.join new = c
c.join insert(c_1, *e*) = insert(*c*.join c_1, *e*)
implies Associative **with** [.join **for** ◯]
converts [.join]

ElementEquality: **trait**
imports Equality **with** [E **for** T]

Member: **trait**
assumes Container, ElementEquality
introduces $\# \in \#$: E, C → Bool
constrains ∈, insert **so that for all** [*c:* C, *e*, e_1: E]
$\neg(e \in \text{new})$
$e \in \text{insert}(c, e_1) = (e = e_1) \mid (e \in c)$
implies converts [∈]

ElemCount: **trait**
assumes Container, ElementEquality
imports Cardinal
introduces count: C, E → Card
constrains count, insert **so that for all** [*e*, e_1: E, *c:* C]
count(new, *e*) = 0
count(insert(*c*, *e*), e_1) = count(*c*, *e*) + (**if** $e = e_1$ **then** 1 **else** 0)
implies converts [count]

Delete: **trait**
Assumes Container
introduces delete: C, E → C
constrains delete **so that for all** [*e:* E]
delete(new, *e*) = new

Containment: **trait**
assumes Container
includes PartiallyOrdered
with [⊂ **for** <, ⊃ **for** >, ⊆ **for** ≤, ⊇ **for** ≥, C **for** T]
constrains C **so that for all** [*e:* E, *c:* C]
$c \subseteq \text{insert}(c, e)$
implies for all [*c:* C]
$\text{new} \subseteq c$

Next: **trait**
assumes Container
introduces next: C → E
constrains next, insert **so that for all** [*e:* E]
next(insert(new, *e*)) = *e*
exempts next(new)

Rest: **trait**
assumes Container
introduces rest: C → C
constrains rest, insert **so that for all** [*e:* E]
rest(insert(new, *e*)) = new
exempts rest(new)

Remainder: **trait**
assumes Container, Rest
imports Cardinal
introduces remainder: C, Card → C
constrains remainder **so that for all** [*c:* C, *i:* Card]
remainder(*c*, 0) = *c*
remainder(*c*, *i* + 1) = remainder(rest(*c*), *i*)
implies converts [remainder]

Index: **trait**
assumes Container, Next, Rest
imports Cardinal
introduces #[#]: C, Card → E
constrains #[#] **so that for all** [*c:* C, *i:* Card]
c[1] = next(*c*)
c[(*i* + 1)] = rest(*c*)[*i*]
implies converts [#[#]]
exempts for all [*c:* C] *c*[0]

Container classes

SetBasics: **trait**
assumes ElementEquality, Container **with** [{} **for** new]
includes Size **with** [{} **for** new], Member **with** [{} **for** new]
introduces delete: C, E → C
constrains C **so that**
C **partitioned by** [∈]
for all [*s:* C, *e*, e_1: E]
size(inserts(*s*, *e*)) = size(*s*) + (**if** *e* ∈ *s* **then** 0 **else** 1)
e_1 ∈ delete(*s*, *e*) = (e_1 ∈ *s*) & (¬(*e* = e_1))
implies Delete **with** [{} **for** new]
converts [size, delete, ∈]

BagBasics: **trait**
assumes ElementEquality, Container **with** [{} **for** new]
imports AdditiveSize **with** [{} **for** new],
ElemCount **with** [{} **for** new]
includes Member **with** [{} **for** new]
introduces delete: C, E → C
constrains C **so that**
C **partitioned by** [count]
for all [*b:* C, *e*, e_1: E]
count(delete(*b*, *e*), e_1) = count(*b*, e_1) − (**if** *e* = e_1 **then** 1 **else** 0)

implies Delete **with** [{} **for** new]
converts [size, delete, count, $\in$]

CollectionExtensions: **trait**
assumes ElementEquality, Container **with** [{} **for** new]
imports IsEmpty **with** [{} **for** new],
Singleton **with** [{} **for** new, {#} **for** singleton],
Containment **with** [{} **for** new],
Join **with** [{} **for** new, $\cup$ **for** .join]
includes Equality **with** [C **for** T]
implies converts [{#}, isEmpty, $\cup$]

SetIntersection: **trait**
assumes SetBasics
introduces $\# \cap \#$: C, C $\rightarrow$ C
constrains $\cap$ **so that for all** [s, s_1: C, e: E]
$e \in (s \cap s_1) = (e \in s)\ \&\ (e \in s_1)$
converts [$\cap$]

Set: **trait**
assumes ElementEquality
imports SetBasics, SetIntersection
includes CollectionExtensions
implies Abelian **with** [$\cup$ **for** $\bigcirc$, C **for** T],
Abelian **with** [$\cap$ **for** $\bigcirc$, C **for** T]
converts [size, delete, $\in$, $\cap$, $\cup$, {#}, isEmpty, =, $\subset$, $\supset$, $\subseteq$, $\supseteq$]

Bag: **trait**
assumes ElementEquality
imports BagBasics
includes CollectionExtensions
implies Abelian **with** [$\cup$ **for** $\bigcirc$, C **for** T]
converts [size, delete, count, $\in$, $\cup$, {#}, isEmpty, =, $\subset$, $\supset$, $\subseteq$, $\supseteq$]

Enumerable: **trait**
imports IsEmpty, Next, Rest
includes Container
constrains C **so that** C **partitioned by** [next, rest, isEmpty]

StackOrQueue: **trait** % For assuming 'Stack or Queue'
includes Enumerable
introduces isStack: $\rightarrow$Bool
constrains next, rest, insert **so that for all** [c: C, e: E]
next(insert(c, e)) = **if** isEmpty(c)|isStack **then** e **else** next(c)
rest(insert(c, e)) = **if** isEmpty(c)|isStack **then** c **else** insert(rest(c), e)
implies converts [next, rest]

Stack: **trait**
includes Enumerable **with** [push **for** insert, top **for** next, pop **for** rest]
constrains push, pop, top **so that for all** [*stk:* C, *e:* E]
top(push(*stk*, *e*)) = *e*
pop(push(*stk*, *e*)) = *stk*
implies StackOrQueue **with** [push **for** insert, top **for** next, pop **for** rest, true **for** isStack]

Queue: **trait**
includes Enumerable **with** [first **for** next]
constrains first, rest, insert **so that for all** [*q:* C, *e:* E]
first(insert(*q*, *e*)) = **if** isEmpty(*q*) **then** *e* **else** first(*q*)
rest(insert(*q*, *e*)) = **if** isEmpty(*q*) **then** new **else** insert(rest(*q*), *e*)
implies StackOrQueue **with** [first **for** next, false **for** isStack]

Dequeue: **trait**
includes Stack **with** [insert **for** push, first **for** top, rest, **for** pop],
Stack **with** [enter **for** push, last **for** top, prefix **for** pop]
constrains C **so that for all** [*c:* C, *e*, e_1: E]
insert(new, *e*) = enter(new, *e*)
insert(enter(*c*, *e*), e_1) = enter(insert(*c*, e_1), *e*)
implies Queue, Queue **with** [enter **for** insert, last **for** first, prefix **for** rest]
converts [insert, first, last, rest, prefix], [enter, first, last, rest, prefix]

Sequence: **trait**
imports Dequeue, AdditiveSize
includes Index **with** [first **for** next],
Join **with** [|| **for** .join]
implies C **partitioned by** [size, #[#]]

SubSequence: **trait**
imports Sequence
includes Remainder **with** [#[#...] **for** remainder],
Remainder **with** [#[...#] **for** remainder, prefix **for** rest]

PriorityQueue: **trait**
assumes TotalOrder **with** [E **for** T]
includes Enumerable
constrains next, rest, insert **so that for all** [*q:* C, *e:* E]
next(insert(*q*, *e*)) = **if** isEmpty(*q*) **then** *e*
else if next(*q*) ≤ *e* **then** next(*q*) **else** *e*
rest(insert(*q*, *e*)) = **if** isEmpty(*q*) **then** new
else if next(*q*) ≤ *e* **then** insert(rest(*q*), *e*) **else** *q*
implies converts [next, rest, isEmpty]

Generic operators on containers

CoerceContainer: **trait**
assumes container **with** [DC **for** C],
Container **with** [RC **for** C]
introduces coerce: DC → RC
constrains coerce **so that for all** [*dc:* DC, *e:* E]
coerce(new) = new
coerce(insert(*dc*, *e*)) = insert(coerce(*dc*), *e*)
implies converts [coerce]

Reduce: **trait**
assumes enumerable, RightIdentity **with** [E **for** T]
introduces reduce: C → E
constrains reduce **so that for all** [*c:* C]
reduce(*c*) = **if** isEmpty(*c*) **then** unit **else** next(*c*) ○ reduce(rest(*c*))
implies converts [reduce]

SomePass: **trait**
assumes Container
introduces
test: E, T → Bool
somePass: C, T → Bool
constrains somePass **so that for all** [*c:* C, *e:* E, *t:* T]
¬somePass(new, *t*)
somePass(insert(*c*, *e*), *t*) = test(*e*, *t*) | somePass(*c*, *t*)
implies converts [somePass]

AllPass: **trait**
assumes container
introduces
test: E, T → Bool
allPass: C, T → Bool
constrains allPass **so that for all** [*c:* C, *e: E*, *t:* T]
allPass(new, *t*)
allPass(insert(*c*, *e*), *t*) = test(*e*, *t*) & allPass(*c*, *t*)
implies converts [allPass]

Sift: **trait**
assumes container
introduces
test: E, T → Bool
sift: C, T → C
constrains sift **so that for all** [*c:* C, *e:* E, *t:* T]
sift(new, *t*) = new

sift(insert(*c*, *e*), *t*) = **if** test(*e*, *t*) **then** insert(sift(*c*, *t*), *e*) **else** **sift**(*c*, *t*)
implies converts [sift]

PairwiseExtension: **trait**
assumes Enumerable
introduces
extOp: C, C→C
elemOp: E, E→E
constrains extOp **so that for all** [c_1, c_2: C, e_1, e_2: E]
extOp(new, new) = new
extOp(insert(c_1, e_1), insert(c_2, e_2))
= insert(extOp(c_1, c_2), elemOp(e_1, e_2))
implies converts [extOp]
exempts for all [*c*: C, *e*: E]
extOp(new, insert(*c*, *e*)),
extOp(insert(*c*, *e*), new)

PointwiseImage: **trait**
assumes Container with [DC **for** C, DE **for** E],
Container **with** [RC **for** C, RE **for** E]
introduces
extOp: DC→RC
pointOp: DE→RE
constrains extOp **so that for all** [*dc*: DC, *de*: DE]
extOp(new) = new
extOp(insert(*dc*, *de*)) = insert(extOp(*dc*), pointOp(*de*))
implies converts [extOp]

Nonlinear structures

BinaryTree: **trait**
imports Cardinal
introduces
⟨#⟩: E→C
⟨#, #⟩: C, C→C
#.left: C→C
#.right: C→C
size: C→Card
isLeaf: C→Bool
content: C→E
constrains C **so that**
C **generated by** [⟨#⟩, ⟨#, #⟩]
C **partitioned by** [.left, .right, content, isLeaf]

```
    for all [tl, tr: C, e: E]
      (⟨tl, tr⟩).left = tl
      (⟨tl, tr⟩).right = tr
      size(⟨e⟩) = 1
      size(⟨tl, tr⟩) = size(tl) + size(tr)
      isLeaf(⟨e⟩)
      ¬isLeaf(⟨tl, tr⟩)
      content(⟨e⟩) = e
  implies for all [t: C] isLeaf(t) = (size(t) = 1)
  converts [.left, .right, size, isLeaf, content]
  exempts for all [tl, tr: C, e: E](⟨e⟩).left, (⟨e⟩).right, content(⟨tl, tr⟩)

BasicGraph: trait
  assumes Equality with [Node for T]
  imports Set with [NodeSet for C, Node for E],
    Pair with [Edge for C, Node for T1, Node for T2]
  introduces
    empty: →Graph
    addNode: Graph, Node→Graph
    addEdge: Graph, Edge→Graph
    nodes: Graph→NodeSet
    adj: Node, Graph→NodeSet
  constrains Graph so that
    Graph generated by [empty, addNode, addEdge]
    Graph partitioned by [nodes, adj]
    for all [g: Graph, e: Edge, n, n1: Node]
      nodes(empty) = {}
      nodes(addNode(g, n)) = insert(nodes(g), n)
      nodes(addEdge(g, e)) = insert(insert(nodes(g), e.first), e.second)
      adj(n, empty) = {}
      adj(n,addNode(g, n1)) = adj(n, g)
      adj(n, addEdge(g, e)) =
        if n = (e.first) then insert(adj(n, g), e.second) else adj(n, g)
  implies converts [nodes, adj]

Connectivity: trait
  assumes Equality with [Node for T], BasicGraph
  introduces
    reach: NodeSet, Graph→NodeSet
    allReach: NodeSet, NodeSet, Graph→Bool
    connected: Graph→Bool
  constrains reach, allReach, connected so that
    for all [g: Graph, e: Edge, ns, ns1: NodeSet, n: Node]
      reach(ns, empty) = {}
```

```
        reach(ns, addNode(g, n)) = reach(ns, g)
        allReach({}, ns, g)
        allReach(insert(ns, n), ns1, g) =
          allReach(ns, ns1, g) & (ns1 ⊆ reach({n}, g))
        connected(g) = allReach(nodes(g), nodes(g), g)
    implies converts [allReach, connected]

Graph: trait
    assumes Equality with [Node for T]
    imports BasicGraph
    includes Connectivity,
      Connectivity with [stronglyConnected for
                           connected, pathReach for reach,
                           allPathReach for AllReach]
    constrains reach, allReach, connected so that
      for all [g: Graph, e: Edge, ns: NodeSet]
        reach(ns, addEdge(g, e)) =
          reach(ns, g) ∪
          (if (e.first) ∈ ns
             then insert(reach({(e.second)}, g), (e.second))
             else if (e.second) ∈ ns
               then insert(reach({(e.first)}, g), (e.first))
               else {})
    constrains pathReach, allPathReach, stronglyConnected so that
      for all [g: Graph, e: Edge, ns: NodeSet]
        pathReach(ns, addEdge(g, e)) =
          pathReach(ns, g) ∪
          (if (e.first) ∈ ns
             then insert(pathReach({(e.second)}, g), (e.second))
             else {})
    implies converts [reach, allReach, connected, pathReach, allPathReach,
                      stronglyConnected]
```

Rings, fields, and numbers

```
Ring: trait
    includes AbelianGroup with [+ for ◯, 0 for unit, −# for inv],
      Semigroup with [* for ◯],
      Distributive
```

RingWithUnit: **trait**
includes Ring, Identity **with** [$*$ **for** $\bigcirc$, 1 **for** unit]

InfixInverse: **trait**
assumes Inverse
introduces $\# \oslash \#$: T, T→T
constrains $\# \oslash \#$ **so that for all** [x, y: T]
$x \oslash y = x \bigcirc \text{inv}(y)$
implies converts [$\# \oslash \#$]

Integer: **trait**
includes RingWithUnit **with** [Int **for** T],
Ordered **with** [Int **for** T],
InfixInverse **with** [+ **for** $\bigcirc$, $-\#$ **for** inv, − **for** $\oslash$, Int **for** T]
asserts Int **generated by** [1, +, $-\#$]
for all [x: Int]
$x < (x+1)$
converts [0, $*$, $\# - \#$, $=$, $\leqslant$, $\geqslant$, $<$, $>$]

Field: **trait**
includes RingWithUnit
introduces $\#^{-1}$: T→T
constrains $*$, $^{-1}$ **so that for all** [x: T]
$(x = 0) \,|\, ((x*(x^{-1})) = 1)$
exempts 0^{-1}

Rational: **trait**
includes Field **with** [R **for** T],
Ordered **with** [R **for** T],
InfixInverse **with** [+ **for** $\bigcirc$, $-\#$ **for** inv, − **for** $\oslash$, R **for** T]
InfixInverse **with** [$*$ **for** $\bigcirc$, $\#^{-1}$ **for** inv, / **for** $\oslash$, R **for** T]
asserts
R **generated by** [1, +, $-\#$, $^{-1}$]
for all [x, y, z: R]
$0 < 1$
$((x+z) < (y+z)) = (x < y)$
$(x = 0) \,|\, ((0 < (x^{-1})) = (0 < x))$
implies converts [0, $*$, $\# - \#$, /, $=$, $\leqslant$, $\geqslant$, $<$, $>$]

Lattices

ExtremalBound: **trait**
assumes PartialOrder
includes AbelianSemigroup **with** [.glb **for** $\bigcirc$]

constrains .glb **so that for all** [*x*, *y*, *z*: T]
$(x.\text{glb}\ y) \leq x$
$((z \leq x)\ \&\ (z \leq y)) \Rightarrow (z \leq (x.\text{glb}\ y))$

Semilattice: **trait**
includes PartiallyOrdered,
ExtremalBound,
ExtremalBound **with** [$\geq$ **for** $\leq$, .lub **for** .glb]
introduces $\bot$: $\rightarrow$T
constrains $\bot$ **so that for all** [*x*: T]
$x \geq \bot$
implies AbelianMonoid **with** [$\bot$ **for** unit, .lub **for** $\bigcirc$]

Lattice: **trait**
includes Semilattice
introduces $\top$: $\rightarrow$T
constraints $\top$ **so that for all** [*x*: T]
$x \leq \top$
implies Lattice **with** [$\top$ **for** $\bot$, $\bot$ **for** $\top$, .glb **for** .lub, .lub **for** .glb,
$\geq$ **for** $\leq$, $\leq$ **for** $\geq$, $>$ **for** $<$, $<$ **for** $>$]

Enumerated data types

Enumerated: **trait**
imports Ordinal
includes Ordered
introduces
first: $\rightarrow$T
last: $\rightarrow$T
succ: T $\rightarrow$ T
pred: T $\rightarrow$ T
ord: T $\rightarrow$ Ord
asserts T **generated by** [first, succ]
T **partitioned by** [ord]
for all [*x*, *y*: T]
ord(first) = first
ord(succ(*x*)) = **if** *x* = last **then** ord(last) **else** succ(ord(*x*))
pred(succ(*x*)) = **if** *x* = last **then** pred(last) **else** *x*
$x \leq y = \text{ord}(x) \leq \text{ord}(y)$
implies T **generated by** [last, pred]
for all [*x*: T]
$(x = \text{first}) \mid (\text{succ}(\text{pred}(x)) = x)$

```
        first ≤ x
        x ≤ last
    converts [=, ≤, ≥, <, >]

Rainbow: trait
  includes Enumerated with [Color for T]
  introduces
    red: →Color
    orange: →Color
    yellow: →Color
    green: →Color
    blue: →Color
    violet: →Color
  asserts
    Color generated by [red, orange, yellow, green, blue, violet]
    first = red
    last = violet
    succ(red) = orange
    succ(orange) = yellow
    succ(yellow) = green
    succ(green) = blue
    succ(blue) = violet
  implies converts [pred, last, ord, =, ≤, ≥, <, >, red, orange,
                    yellow, green, blue, violet, violet],
```

Display traits

% The following traits represented a fairly straightforward translation of the specifications in "Formal specification as a design tool" [2]. We have not attempted to improve the design presented there, merely to translate it into Larch.

```
Coordinate: trait
  introduces minus: Coordinate, Coordinate → Coordinate

Illumination: trait
  introduces combine: Illumination, Illumination → Illumination

Boundary: trait
  introduces apply: Boundary, Coordinate → Bool

Transform: trait
  introduces apply: Transformation, Coordinate → Coordinate
```

```
Displayable: trait
  introduces
    appearance: T, Coordinate → Illumination
    in: T, Coordinate → Bool


Picture: trait
  assumes Boundary, Transform, Illumination,
    Displayable with [Contents for T]
  includes Displayable with [Picture for T]
  introduces makePicture: Contents, Boundary, Transformation → Picture
  constrains Picture so that
    Picture generated by [makePicture]
    for all [cn: Contents, b: Boundary, t: Transformation,
             cd: Coordinate]
      appearance(makePicture(cn, b, t), cd) = appearance(cn, apply(t, cd))
      in(makePicture(cn, b, t), cd) = apply(b, cd)
  implies converts [appearance: Picture, Coordinate → Illumination,
                    in: Picture, Coordinate → Bool]


Contents: trait
  assumes Coordinate, Illumination, Displayable with
    [Component for T]
  includes Displayable with [Contents for T]
  introduces
    empty: → Contents
    addComponent: Contents, Component, Coordinate → Contents
  constrains Contents so that
    Contents generated by [empty, addComponent]
    for all [cn: contents, cm: component, cd, cd1: Coordinate]
      appearance(addComponent(cn, cm, cd1, cd)) =
        if in(cm, minus(cd, cd1))
        then (if in(cn, cd)
          then combine(appearance(cm, minus(cd, cd1)),
                       appearance(cn, cd))
          else appearance(cm, minus(cd, cd1)))
        else appearance(cn, cd)
      ¬in(empty, cd)
      in(add(Component(cn, cm, cd1), cd) =
        in(cm, minus(cd, cd1))|in(cn, cd)
  implies converts [appearance: Contents, Coordinate → Illumination,
                    in: Contents, Coordinate → Bool]
```

```
    exempts for all [cd: Coordinate] appearance(empty, cd)

Component: trait
  assumes Displayable with [View for T],
    Displayable with [Text for T],
    Displayable with [Figure for T]
  includes componentCoercion with [View for T, coerceView for coerce],
    ComponentCoercion with [Text for T, coerceText for coerce],
    ComponentCoercion with [Figure for T, coerceFigure for coerce]

ComponentCoercion: trait
  assumes Displayable
  includes Displayable with [Component for T]
  introduces coerce: T→Component
  constrains Components so that for all [t: T, cd: Coordinate]
    appearance(coerce(t), cd) = appearance(t, cd)
    in(coerce(t), cd) = in(t, cd)

View: trait
  assumes Displayable with [Picture for T],
    Equality with [PictureId for T],
    Container with [IdList for C, PictureId for E],
    Coordinate
  includes Displayable with [View for T]
  introduces
    empty: →View
    addPicture: View, Coordinate, PictureId, Picture→View
    findPictures: View, Coordinate→IdList
    deletePicture: View, PictureId→View
  constrains View so that
    View generated by [empty, addPicture]
    for all [v: View, cd, cd1: Coordinate, id, id1: PictureId, p: Picture]
    appearance(addPicture(v, cd1, id, p), cd) =
      if in(p, minus(cd, cd1))
        then appearance(p, minus(cd, cd1))
        else appearance(v, cd)
    ¬in(empty, cd)
    in(addPicture(v, cd1, id, p), cd) = (in(p, minus(cd, cd1))|in(v, cd))
    findPictures(empty, cd) = new
    findPictures(addPicture(v, cd1, id, p), cd) =
      if in(p, minus(cd, cd1))
        then insert(id, findPictures(v, cd))
        else findPictures(v, cd)
      deletePicture(empty, id) = empty
```

```
        deletePicture(addPicture(v, cd1, id1, p), id) =
        if id = id1 then v else addPicture(deletePicture(v, id),
          cd, id1, p)
    implies converts [findPictures, deletePicture,
                      appearance:View, Coordinate → Illumination,
                      in:View, Coordinate → Bool]
    exempts for all [cd: Coordinate] appearance(empty, cd)

Display: trait
    assumes Boundary, Transform, Illumination, Coordinate,
      Equality with [PictureId for T],
      Container with [IdList for C, PictureId for E]
    includes Picture, Contents, Component, View
```

Acknowledgment

Ron Kownacki, Greg Nelson, Mary Shaw and Jeannette Wing all helped us with this set of examples. Greg was particularly helpful with the more algebraic examples, and Mary with the sections on 'containers'. Jeannette and Ron went over all the examples rather carefully, and were responsible for a number of improvements.

References

[1] J. Abrial, The specification language Z: Syntax and semantics, Oxford University Computing Laboratory, Programming Research Group, Oxford, 1980.

[2] J.V. Guttag and J.J. Horning, Formal specification as a design tool, *Proc. ACM Symposium on Principles of Programming Languages*, Las Vegas (1980) 251-261.

[3] J.V. Guttag and J.J. Horning, Report on the Larch Shared Language, *Sci. Comput. Programming* **6** (1986) 103-134.